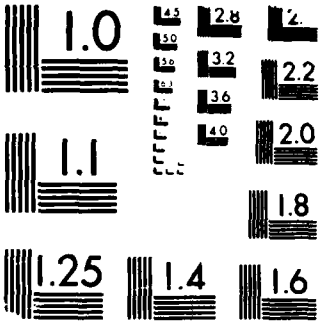
AD-A194 034 IMPROVED TIME BOUNDS FOR THE MAXIMUM FLOW PROBLEM(U) 1/1
PRINCETON UNIV NJ DEPT OF COMPUTER SCIENCE
R K AHUMA ET AL. SEP 87 CS-TR-118-87 N00014-87-K-0467
UNCLASSIFIED F/G 12/4 NL



Princeton University

IMPROVED TIME BOUNDS FOR THE MAXIMUM FLOW PROBLEM

Ravindra K. Ahuja
James B. Orlin
Robert E. Tarjan

CS-TR-118-87

September 1987

Department of Computer Science

(4)

IMPROVED TIME BOUNDS FOR THE MAXIMUM FLOW PROBLEM

Ravindra K. Ahuja
James B. Orlin
Robert E. Tarjan

CS-TR-118-87

September 1987

Improved Time Bounds for the Maximum Flow Problem

Ravindra K. Ahuja [1,2]

James B. Orlin [1]

Robert E. Tarjan [3]

September, 1987

ABSTRACT

Recently, Goldberg proposed a new approach to the maximum network flow problem. The approach yields a very simple algorithm running in $O(n^3)$ time on n-vertex networks. Incorporation of the dynamic tree data structure of Sleator and Tarjan yields a more complicated algorithm with a running time of $O(nm \log (n^2/m))$ on m-edge networks. Ahuja and Orlin developed a variant of Goldberg's algorithm that uses scaling and runs in $O(nm + n^2 \log U)$ time on networks with integer edge capacities bounded by U. In this paper we first obtain a modification of the Ahuja-Orlin algorithm with a running time of $O(nm + n^2 \frac{\log U}{\log\log U})$. We then show that the use of dynamic trees in this algorithm further reduces the time bound to $O(nm \log(\frac{n \log U}{m \log\log U} + 2))$. This result demonstrates that the combined use of scaling and dynamic trees results in speed not obtained by using either technique alone.

[1] Sloan School of Management, M.I.T., Cambridge, MA 02139. Research partially supported by a Presidential Young Investigator Fellowship from the NSF, Contract 8451517 ECS, and grants from Analog Devices, Apple Computer Inc., and Prime Computer.

[2] On leave from the Indian Institute of Technology, Kanpur, India.

[3] Dept. of C. S., Princeton University, Princeton, NJ 08544 and AT&T Bell Labs, Murray Hill, NJ 07974. Research partially supported by NSF Grant DCR-8605962 and Office of Naval Research Contract N00014-87-K-0467.

Improved Time Bounds for the Maximum Flow Problem

Ravindra K. Ahuja [1,2]

James B. Orlin [1]

Robert E. Tarjan [3]

September, 1987

1. Introduction

We consider algorithms for the classical maximum network flow problem [4,5,11,13,18]. We formulate the problem as follows. Let $G = (V,E)$ be a directed graph with vertex set V and edge set E. The graph G is a *flow network* if it has two distinct distinguished vertices, a *sink* s and a *source* t, and a non-negative real-valued *capacity* $u(v,w)$ on each edge $(v,w) \in E$. We assume that G is *symmetric*, i.e. $(v,w) \in E$ iff $(w,v) \in E$. We denote by n,m, and U the number of vertices, the number of edges, and the maximum edge capacity, respectively. For ease in stating time bounds, we assume $m \geq n-1$ and $U \geq 4$. Bounds containing U are subject to the assumption that all edge capacities are integral. All logarithms in the paper are base two unless an explicit base is given.

A *flow* f on a network G is a real-valued function f on the edges satisfying the following constraints:

$$f(v,w) \leq u(v,w) \text{ for all } (v,w) \in E \text{ (capacity constraint),} \tag{1}$$

$$f(v,w) = -f(w,v) \text{ for all } (v,w) \in E \text{ (antisymmetry constraint),} \tag{2}$$

$$\sum_{v:(v,w) \in E} f(v,w) = 0 \text{ for all } w \in V-\{s,t\} \text{ (conservation constraint).} \tag{3}$$

[1] Sloan School of Management, M.I.T., Cambridge, MA 02139. Research partially supported by a Presidential Young Investigator Fellowship from the NSF, Contract 8451517 ECS, and grants from Analog Devices, Apple Computer Inc., and Prime Computer.

[2] On leave from the Indian Institute of Technology, Kanpur, India.

[3] Dept. of C. S., Princeton University, Princeton, NJ 08544 and AT&T Bell Labs, Murray Hill, NJ 07974. Research partially supported by NSF Grant DCR-8605962 and Office of Naval Research Contract N00014-87-K-0467.

The *value* $|f|$ of a flow f is the net flow into the sink:

$$|f| = \sum_{v:(v,t) \in E} f(v,t).$$

A *maximum flow* is a flow of maximum value. The *maximum flow problem* is that of finding a maximum flow in a given network.

Remark. We assume that all edge capacities are finite. If some edge capacities are infinite but no path of infinite-capacity edges from s to t exists, then each infinite capacity can be replaced by the sum of the finite capacities, without affecting the problem. □

The maximum flow problem has a long and rich history, and a series of faster and faster algorithms for the problem have been developed. (See [9] for a brief survey.) The previously fastest known algorithms are those of Goldberg and Tarjan [7,9], with a running time of $O(nm \log (n^2/m))$, and that of Ahuja and Orlin [1], with a running time of $O(nm + n^2 \log U)$. Both of these algorithms are refinements of a generic method proposed by Goldberg [7], which we shall call the *preflow algorithm*. For networks with $m = \Omega(n^2)$, the Goldberg-Tarjan bound is $O(n^3)$, which matches the bound of several earlier algorithms [10,12,14,19]. For networks with $m = O(n^{2-\epsilon})$ for some constant $\epsilon > 0$, the Goldberg-Tarjan bound is $O(nm \log n)$, which matches the bound of the earlier Sleator-Tarjan algorithm [15,16]. Under the *similarity assumption* [6], namely $U = O(n^k)$ for some constant k, the Ahuja-Orlin bound beats the Goldberg-Tarjan bound unless $m = O(n)$ or $m = \Omega(n^2)$.

The Goldberg-Tarjan and Ahuja-Orlin algorithms obtain their speed from two different techniques. The former uses a sophisticated data structure, the dynamic tree structure of Sleator and Tarjan [16,17,18], whereas the latter uses scaling. Our main purpose in this paper is to demonstrate that the use of both techniques results in efficiency not obtained by using either technique alone. We first modify the Ahuja-Orlin algorithm to obtain a small improvement in running time, to $O(nm + n^2 \frac{\log U}{\log\log U})$. Then we show that the use of dynamic trees in the modified algorithm results in a running time of $O(nm \log(\frac{n \log U}{m \log\log U} + 2))$. Under the similarity assumption, this bound is better than all previously known bounds for the maximum flow problem. For networks with $m = O(n)$ and $U = O(n^k)$ for some constant k, the bound is $O(nm \log\log n)$, which beats both the Goldberg-Tarjan bound and the original Ahuja-Orlin bound by a factor of $\log n / \log\log n$. Moreover, the bound rapidly approaches $O(nm)$ as the graph density m/n increases.

Our paper consists of four sections in addition to this introduction. In Section 2 we review the preflow algorithm. In Section 3 we review the Ahuja-Orlin algorithm and describe and analyze our improvement of it. In Section 4 we combine the use of dynamic trees with the method of Section 3 and analyze the resulting algorithm. Section 5 contains some final remarks.

2. The Preflow Algorithm

In contrast to the classical *augmenting path method* of Ford and Fulkerson [5], which moves flow along an entire path from s to t at once, the preflow method moves flow along a single edge at a time. The key concept underlying the algorithm is that of a preflow, introduced by Karzanov [10]. A *preflow* f is a real-valued function on the edges satisfying constants (1), (2), and a relaxation of (3). For any vertex w, let the *flow excess* of w be $e(w) = \sum_{v:(v,w) \in E} f(v,w)$.

The required constraint is the following:

$$e(w) \geq 0 \text{ for all } w \in V-\{s\} \text{ (nonnegativity constraint).} \tag{4}$$

We call a vertex v *active* if $v \neq t$ and $e(v) > 0$. Observe that the nonnegativity constraint implies that $e(s) \leq 0$.

The *residual capacity* of an edge (v,w) with respect to a preflow f is $u_f(v,w) = u(v,w)-f(v,w)$. An edge is *saturated* if $u_f(v,w) = 0$ and *unsaturated* otherwise. (The capacity constraint implies that any unsaturated edge (v,w) has $u_f(v,w) > 0$.)

The preflow algorithm maintains a preflow and moves flow from active vertices through unsaturated edges toward the sink, along paths estimated to contain as few edges as possible. Excess flow that cannot be moved to the sink is returned to the source, also along estimated shortest paths. Eventually the preflow becomes a flow, which is a maximum flow.

As an estimate of path lengths, the algorithm uses a *valid labeling*, which is a function d from the vertices to the nonnegative integers such that $d(s) = n$, $d(t) = 0$, and $d(v) \leq d(w) + 1$ for every unsaturated edge (v,w). A proof by induction shows that, for any valid labeling d, $d(v) \leq \min \{d_f(v,s) + n, d_f(v,t)\}$, where $d_f(v,w)$ is the minimum number of edges on a path from v to w consisting of edges unsaturated with respect to the flow f. We call an edge (v,w) *eligible* if (v,w) is unsaturated and $d(v) = d(w) + 1$.

The algorithm begins with an initial preflow f and valid labeling d defined as follows:

$$f(v,w) = \begin{cases} u(v,w) & \text{if } v = s, \\ -u(w,v) & \text{if } w = s, \\ 0 & \text{if } v \neq s \text{ and } w \neq s, \end{cases}$$

$$d(v) = \min \{d_f(v,s) + n, d_f(v,t)\}.$$

The algorithm consists of repeating the following two steps, in any order, until no vertex is active:

push (v,w).

Applicability: Vertex v is active and edge (v,w) is eligible.

Action: Increase $f(v,w)$ by $\min \{e(v), u_f(v,w)\}$. The push is *saturating* if (v,w) is saturated after the push and *nonsaturating* otherwise.

relabel (v).

Applicability: Vertex v is active and no edge (v,w) is eligible.

Action: Replace $d(v)$ by $\min \{d(w) + 1 \mid (v,w) \text{ is unsaturated}\}$.

When the algorithm terminates, f is a maximum flow. Goldberg and Tarjan derived the following bounds on the number of steps required by the algorithm:

Lemma 2.1 [9]. Relabeling a vertex v strictly increases $d(v)$. No vertex label exceeds $2n-1$. The total number of relabelings is $O(n^2)$.

Lemma 2.2 [9]. There are at most $O(nm)$ saturating pushes and at most $O(n^2m)$ nonsaturating pushes.

Efficient implementations of the above algorithm require a mechanism for selecting pushing and relabeling steps to perform. Goldberg and Tarjan proposed the following method. For each vertex, construct a (fixed) list $A(v)$ of the edges out of v. Designate one of these edges, initially the first on the list, as the *current edge* out of v. To execute the algorithm, repeat the following

step until there no active vertices:

push/relabel (v).

Applicability: Vertex v is active.

Action: If the current edge (v,w) of v is eligible, perform *push*(v,w). Otherwise, if (v,w) is not the last edge on $A(v)$, make the next edge after (v,w) the current one. Otherwise, perform *relabel* (v) and make the first edge on $A(v)$ the current one.

With this implementation, the algorithm runs in $O(nm)$ time plus $O(1)$ time per nonsaturating push. This gives an $O(n^2m)$ time bound for any order of selecting vertices for push/relabel steps. Making the algorithm faster requires reducing the time spent on nonsaturating pushes. The number of such pushes can be reduced by selecting vertices for push/relabel steps carefully. Goldberg and Tarjan showed that FIFO selection (first active, first selected) reduces the number of nonsaturating pushes to $O(n^3)$. Cheriyan and Maheshwari [2] showed that highest label selection (always pushing flow from a vertex with highest label) reduces the number of nonsaturating pushes to $O(n^2m^{1/2})$. Ahuja and Orlin proposed a third selection rule, which we discuss in the next section.

3. The Scaling Preflow Algorithm

The intuitive idea behind the Ahuja-Orlin algorithm, henceforth called the *scaling preflow algorithm*, is to move large amounts of flow when possible. The same idea is behind the maximum capacity augmenting path method of Edmonds and Karp [3] and the capacity scaling algorithm of Gabow [6]. One way to apply this idea to the preflow algorithm is to always push flow from a vertex of large excess to a vertex of small excess, or to the sink. The effect of this is to reduce the maximum excess at a rapid rate.

Making this method precise requires specifying when an excess is large and when small. For this purpose the scaling preflow algorithm uses an *excess bound* Δ and an integer *scaling factor* $k \geq 2$. A vertex v is said to have *large excess* if its excess exceeds Δ/k and *small excess* otherwise. As the algorithm proceeds, k remains fixed, but Δ periodically decreases. Initially, Δ is the smallest power of k such that $\Delta \geq U$. The algorithm maintains the invariant that $e(v) \leq \Delta$ for every active vertex v. This requires changing the pushing step to the following.

push (v,w).

Applicability: Vertex v is active and edge (v,w) is eligible.

Action: If $w \neq t$, increase $f(v,w)$ by min $\{e(v), u_f(v,w), \Delta - e(w)\}$. Otherwise ($w = t$),

increase $f(v,w)$ by $\min \{ e(v), u_f(v,w) \}$.

The algorithm consists of a number of *scaling phases*, during each of which Δ remains constant. A phase consists of repeating push/relabel steps, using the following selection rule, until no active vertex has large excess, and then replacing Δ by Δ/k.

Large excess, smallest label selection: Apply a push/relabel step to a vertex v of large excess; among such vertices, choose one of smallest label.

If the edge capacities are integers, the algorithm terminates after at most $\lfloor \log_k U + 1 \rfloor$ phases: after $\lfloor \log_k U + 1 \rfloor$ phases, $\Delta < 1$, which implies that f is a flow, since the algorithm maintains integrality of excesses. Ahuja and Orlin derived a bound of $O(kn^2 \log_k U)$ on the total number of nonsaturating pushes. We repeat the analysis here, since it provides motivation for our modification of the algorithm.

Lemma 3.1 [1]. The total number of nonsaturating pushes in the scaling preflow algorithm is $O(kn^2 (\log_k U + 1))$.

Proof. Consider the function $\Phi = \sum_{v \text{ active}} e(v)\, d(v)/\Delta$. We call Φ the *potential* of the current preflow f and labeling d. Since $0 < e(v)/\Delta \leq 1$ and $0 \leq d(v) \leq 2n$ for every active vertex v, $0 \leq \Phi \leq 2n^2$ throughout the algorithm. Every pushing step decreases Φ. A nonsaturating pushing step decreases Φ by at least $1/k$, since the push is from a vertex v with excess more than Δ/k to a vertex w with $d(w) = d(v) - 1$, and $e(w) \leq \Delta/k$ or $w = t$. The value of Φ can increase only during a relabeling or when Δ changes. A relabeling of a vertex v increases Φ by at most the amount $d(v)$ increases. Thus the total increase in Δ due to relabelings, over the entire algorithm, is at most $2n^2$. When Δ changes, Φ increases by a factor of k, to at most $2n^2$. This happens at most $\lfloor \log_k U + 1 \rfloor$ times. Thus the total increase in Φ over the entire algorithm is at most $2n^2 \lfloor \log_k U + 2 \rfloor$. The total number of nonsaturating pushes is at most k times the sum of the initial value of Φ and the total increase in Φ. This is at most $2kn^2 \lfloor \log_k U + 3 \rfloor$. □

Choosing k to be constant independent of n gives a total time bound of $O(nm + n^2 \log U)$ for the scaling preflow algorithm, given an efficient implementation of the vertex selection rule. One way to implement the rule is to maintain an array of sets indexed by vertex label, each set containing all large excess vertices with the corresponding label, and to maintain a pointer to the nonempty set of smallest index. The total time needed to maintain this structure is $O(nm + n^2 \log U)$.

Remark. The bound for the scaling preflow algorithm can be improved slightly if it is measured in terms of a different parameter. Let $U^* = 4 + \sum_{v:(s,v) \in E} u(s,v)/n$. Then the bound on nonsaturating pushes can be reduced to $O(n^2 \log U^*)$ and the overall time bound to $O(nm + n^2 \log U^*)$. This improvement is analogous to the bound Edmonds and Karp derived for their capacity-scaling transportation algorithm [3]. The argument is as follows. The algorithm maintains the invariant that the total excess on active vertices is at most nU^*. Let phase i be the first phase such that $\Delta \le U^*$. Then the total increase in Φ due to phase changes up to and including the change from phase $i-1$ to phase i is at most $n \sum_{j=0}^{i} 2n/2^{i-j} = O(n^2)$. The total increase in Φ due to later phase changes is $O(n^2 \log U^*)$. □

Having described the Ahuja-Orlin algorithm, we consider the question of whether its running time can be improved by reducing the number of nonsaturating pushes. The proof of Lemma 3.1 bounds the number of nonsaturating pushes by estimating the total increase in the potential Φ. Observe that there is an imbalance in this estimate: $O(n^2 \log_k U)$ of the increase is due to phase changes, whereas only $O(n^2)$ is due to relabelings. Our plan is to improve this estimate by decreasing the contribution of the phase changes, at the cost of increasing the contribution of the relabelings. Making this plan work requires changing the algorithm.

We use a larger scale factor and a slightly more elaborate method of vertex selection. We push flow from a vertex of large excess, choosing among such vertices one with highest label. This vertex selection rule does not guarantee that a nonsaturating push moves enough flow, and we need a stack-based implementation to ensure this. The algorithm maintains a stack S of vertices, with the property that if w is just on top of v on the stack, then (v,w) is the current edge out of v. At the beginning of a scaling phase, the stack is initialized to be empty. The phase consists of repeating the following step until no vertex has large excess, and then replacing Δ by Δ/k.

stack push/relabel: If S is empty, push onto S any large-excess active vertex of largest label.

Now S is definitely nonempty. Let v be the top vertex on S and (v,w) the current edge out of v. Apply the appropriate one of the following three cases:

Case 1: (v,w) is eligible. Perform *push* (v,w) (modified as described at the beginning of this section). If the push is nonsaturating and $e(v) > 0$, push w onto S. Otherwise, while S is nonempty and its top vertex has small excess, pop S.

Case 2: (v,w) is not eligible and is not last on $A(v)$. Replace (v,w) as the current edge out of v by the next edge on $A(v)$.

Case 3: (v,w) *is not eligible and is last on* $A(v)$. Relabel v and pop it from S. Replace (v,w) as the current edge out of v by the first edge on $A(v)$. While S is nonempty and its top vertex has small excess, pop S.

Observe that the stack push/relabel step is just a push/relabel step augmented with some manipulation of the stacks.

Before carrying out a detailed analysis of this algorithm, we make several observations. Every push made by the algorithm is from a vertex of large excess. A push along an edge (v,w) results in one of three events: saturation of (v,w), conversion of v into an inactive vertex, or addition of w to S with an excess of Δ. In the third case (and only in the third case), the push may move zero flow.

Lemma 3.2. The total number of nonsaturating pushes made by the stack-based scaling preflow algorithm is $O(nm + kn^2 + n^2 (\log_k U + 1))$.

Proof. To bound the number of nonsaturating pushes, we use an argument like the proof of Lemma 3.1, but with two potentials instead of one. The first potential is that of Lemma 3.1, $\Phi = \sum_{v \text{ active}} e(v)\, d(v)/\Delta$. By the analysis in the proof of Lemma 3.1, every push decreases Φ, and the total increase in Φ over all phases is $O(n^2 \log_k U)$. Every nonsaturating push either increases or decreases the size of S. Consider a nonsaturating push that increases the size of S, say by pushing flow from v to w and stacking w. When w is stacked, it has an excess of Δ. It cannot be unstacked until it is relabeled or until its excess decreases to at most Δ/k. In the former case we charge the push from v to w to the relabeling of w, in the latter, to the at least $1-1/k$ decrease in Φ caused by pushes from w after w is stacked and before it is unstacked. Since $1 - 1/k \geq 1/2$ for all $k \geq 2$, there are a total of $O(n^2)$ pushes charged to relabelings and $O(n^2 \log_k U)$ pushes charged to decreases in Φ.

We now count the nonsaturating pushes that reduce the size of S. The number of such pushes that begin with more than one vertex S is $O(n^2 \log_k U)$, since each such push must be preceded by a nonsaturating push that adds a vertex to S, and the number of these is $O(n^2 \log_k U)$ by the above argument.

The remaining pushes that must be counted are those that begin with one vertex on S and end with none on S. We call such pushes *emptying*. An emptying push can cause a rather small decrease in Φ, namely $1/k$; thus using Φ to bound such pushes only gives a bound of $O(kn^2 \log_k U)$ on their number. We count emptying pushes more carefully by using a second potential, Φ_2. The definition of Φ_2 involves two parameters, an integer l and a set P. The value

of l is equal to the minimum of $2n$ and the smallest label of a vertex added to S while S was empty during the current phase. Observe that $l = 2n$ at the beginning of a phase and l is nonincreasing during a phase. The set P consists of all vertices with label greater than l and all vertices with label equal to l from which an emptying push has been made during the current phase. Observe that P is empty at the beginning of a phase and P never loses a vertex during a phase. The definition of Φ_2 is

$$\Phi_2 = \sum_{v \in P: e(v) > 0} e(v)\,(d(v) - l + 1)/\Delta. \quad (\text{If } P = \varnothing,\ \Phi_2 = 0.)$$

Observe that $0 \le \Phi_2 \le 2n^2$. Any emptying push either adds a vertex to P or decreases Φ_2 by at least $1/k$. The number of vertices added to P is at most $n\lfloor \log_k U + 1 \rfloor$ over all phases, and hence so is the number of emptying pushes that do not decrease Φ_2 by at least $1/k$.

To bound the number of emptying pushes that decrease Φ_2 by at least $1/k$, we bound the total increase in Φ_2. Increases in Φ_2 are due to relabelings and to decreases in l. (A vertex added to P because of an emptying push has zero excess and hence adds nothing to Φ_2.) A relabeling of a vertex v increases Φ_2 by at most the increase in $d(v)$ plus one; the "plus one" accounts for the fact that the relabeling may add v to P. Thus relabelings contribute at most $4n^2$ to the growth of Φ_2.

There are at most $2n$ decreases in l per phase. A decrease in l by one adds at most n/k to Φ_2, since when the decrease occurs every vertex in P has small excess. Thus the total increase in Φ_2 due to decreases in l is at most $2n^2\lfloor \log_k U + 1 \rfloor / k$ over all phases.

The total number of emptying pushes that decrease Φ_2 by at least $1/k$ is at most k times the total increase in Φ_2, since Φ_2 is initially zero. Thus the total number of such pushes is at most $4kn^2 + 2n^2\lfloor \log_k U + 1 \rfloor$. □

As in the Goldberg-Tarjan and Ahuja-Orlin algorithms, the time to perform saturating pushes and relabeling operations is $O(nm)$. The only significant remaining issue is how to choose vertices to add to S when S is empty. For this purpose, we maintain a data structure consisting of a collection of doubly linked lists $list(r) = \{i \in N:\ e(i) > \Delta/k$ and $d(i) = r\}$ for each $r \in \{1,2,\ldots,2n-1\}$. We also maintain a pointer to indicate the largest index r for which $list(r)$ is non-empty. Maintaining this structure requires $O(1)$ time per push operation plus time to maintain the pointer. The pointer can only increase due to a relabeling (by at most the amount of change in label) or due to a phase change (by at most $2n$). Consequently, the number of times the pointer needs to be incremented or decremented is $O(n^2 + n(\log_k U + 1))$. The overall running

time of the algorithm is thus $O(nm + kn^2 + n^2(\log_k U + 1))$. Choosing $k = \lceil \log U / \log\log U \rceil$ gives the following result:

Theorem 3.3. The stack-based scaling preflow algorithm, with an appropriate choice of k, runs in $O(nm + n^2 \frac{\log U}{\log\log U})$ time.

Remark. This bound can be improved to $O(nm + n^2 \log U^* / \log\log U^*)$, where $U^* = 4 + \sum_{v:(s,v) \in E} u(s,v)/n$. The algorithm must be changed slightly. We use a scaling factor of $k = 2$ until $\Delta \leq U^*$ and then switch to $k = \lceil \log U^* / \log\log U^* \rceil$ for the remainder of the algorithm. When the switch occurs, we increase Δ (if necessary) to be a power of k. The analysis of the initial phases (those in which $\Delta > U^*$) is the same as that in the remark following Lemma 3.1. These phases account for only $O(n^2)$ nonsaturating pushes. The analysis of the remaining phases is as above with U replaced by U^*. □

4. Use of Dynamic Trees

The approach taken in Section 3 was to reduce the total number of pushes by choosing vertices for push/relabel steps carefully. An orthogonal approach is to reduce the total time of the pushes without necessarily reducing their number. This can be done by using the dynamic tree data structure of Sleator and Tarjan [16,17,18]. We conjecture that, given a version of the preflow algorithm with a bound of $p \geq mn$ on the total number of pushes, the running time can be reduced from $O(p)$ to $O(nm \log(\frac{p}{mn} + 1))$ by using dynamic trees. Although we do not know how to prove a general theorem to this effect, we have been able to obtain such a result for each version of the preflow algorithm we have considered. As an example, the $O(nm \log(n^2/m))$ time bound of Goldberg and Tarjan results from using dynamic trees with the FIFO selection rule; the bound on the number of pushes in this case is $O(n^3)$. In this section we shall show that the same idea applies to the scaling preflow algorithm of Section 3, resulting in a time bound of $O(nm \log(\frac{n \log U}{m \log\log U} + 2))$.

The dynamic tree data structure allows the maintenance of a collection of vertex-disjoint rooted trees, each edge of which has an associated real value. Each tree edge is regarded as being directed from child to parent. We regard every vertex as being both an ancestor and a descendant of itself. The data structure supports the following seven operations:

find root (v): Find and return the root of the tree containing vertex v.

find size (v): Find and return the number of vertices in the tree containing vertex v.

find value (v): Find and return the value of the tree edge leaving v. If v is a tree root, the value returned is infinity.

find min (v): Find and return the ancestor w of v with minimum *find value* (w). In case of a tie, choose the vertex w closest to the tree root.

change value (v,x): Add real number x to the value of every edge along the path from v to *find root* (v).

link (v,w,x): Combine the trees containing v and w by making w the parent of v and giving the edge (v,w) the value x. This operation does nothing if v and w are in the same tree or if v is not a tree root.

cut (v): Break the tree containing v into two trees by deleting the edge joining v to its parent; return the value of the deleted edge. This operation breaks no edge and returns infinity if v is a tree root.

A sequence of l tree operations, starting with an initial collection of singleton trees, takes $O(l\log(z+1))$ time if z is the maximum tree size [16,17,18].

In our application, the dynamic tree edges form a subset of the current edges out of all the vertices. Every dynamic tree edge is an eligible edge; its value is its residual capacity. The dynamic tree data structure allows flow to be moved along an entire path at once, rather than along a single edge at a time. The following version of the preflow algorithm, called the *tree preflow algorithm*, uses this ability. Two parameters govern the behavior of the algorithm, a bound Δ on the maximum excess at an active vertex, and a bound z, $1 < z \leq n$, on the maximum size of a dynamic tree. The algorithm consists of beginning with an initial preflow f and a valid labeling d, and with every vertex as a one-vertex dynamic tree, and repeating the following step until there are no active vertices:

tree push/relabel (v):

Applicability: Vertex v is active.

Action: Let (v,w) be the current edge out of v. Apply the appropriate one of the following cases:

Case 1: (v,w) *is eligible.* Let $x =$ *find root* (w), $\varepsilon = \min \{e(v), u_f(v,w),$ *find min* $(w)\}$, and $\delta = \varepsilon$ if $x = t$, $\delta = \min \{\varepsilon, \Delta - e(x)\}$ if $x \neq t$. Send δ units of flow from v to x by increasing $f(v,w)$ by δ and performing *change value* $(w,-\delta)$. This is called a *tree push* from v to x. The tree push is *saturating* if $\delta = \min \{ u_f(v,w),$ *find min* $(v)\}$ (before the push) and *nonsaturating* otherwise. If the tree push is nonsaturating, $e(v) = 0$ (after the push), and *find size* (v) + *find size* $(w) \leq z$, perform *link* $(v,w,u_f(v,w))$. Otherwise, while *find value* (*find min* $(w)) = 0$ perform *cut* (*find min* (w)).

Case 2: (v,w) *is not eligible and not last on* $A(v)$. Replace (v,w) as the current edge out of v by the next edge on $A(v)$.

Case 3: (v,w) *is not eligible and is last on* $A(v)$. Relabel v. Replace (v,w) as the current edge out of v by the first edge on $A(v)$. For every tree edge (y,v) perform *cut* (y).

This algorithm stores flow in two different ways: explicitly for edges that are not dynamic tree edges and implicitly in the dynamic tree data structure for edges that are dynamic tree edges. After each cut, the flow on the edge cut must be restored to its correct current value. In addition, when the algorithm terminates, the correct flow value on each remaining tree edge must be computed. For edges cut during the computation, the desired flow values are easily computed, since the current residual capacity of an edge that is cut is returned by the cut operation. Computing correct flows on termination can be done using at most n *find value* operations.

The algorithm just presented is a generic, modified version of the dynamic tree algorithm of Goldberg and Tarjan [8,9]. It maintains the following invariants: every active vertex is a tree root; no excess exceeds Δ; no tree size exceeds z; every tree edge is eligible. The analysis of Goldberg and Tarjan gives the following result:

Lemma 4.1 [9]. The total time required by the dynamic tree preflow algorithm is $O(nm \log (z+1))$ plus $O(\log (z+1))$ per tree push. The number of links, cuts, and saturating tree pushes is $O(nm)$.

The efficiency of the algorithm depends on the number of nonsaturating tree pushes, which in turn depends on how tree push/relabel steps are selected. We shall show that a selection order analogous to that used in Section 3 results in a bound of $O(nm + kn^2 + \frac{n^2}{z}(\log_k U + 1))$ on the

number of nonsaturating tree pushes, where k is the scaling factor. From this a total time bound of $O(nm \log (\frac{n \log U}{m \log \log U} + 2))$ follows by an appropriate choice of k and z.

We call the resulting version of the preflow algorithm the *tree/scaling preflow algorithm*. In addition to parameters Δ (the maximum excess) and z (the maximum tree size), the algorithm uses a third parameter k, the scaling factor, and a stack S. Whereas z and k remain fixed throughout the running of the algorithm, Δ decreases. Initially $S = \phi$ and Δ is the smallest integer power of k such that $\Delta \geq U$. An active vertex v is said to have *large excess* if $e(v) > \Delta/k$ and *small excess* otherwise. A dynamic tree is said to be *large* if it contains more than $z/2$ vertices and *small* otherwise. The tree/scaling algorithm consists of repeatedly applying the tree push/relabel step to the top vertex on S, while manipulating S according to the following rules:

Rule 1. If S is empty, add to S any active vertex that is the root of a small tree. If all active vertices are roots of large trees, add to S a large-excess active vertex of largest label. If all active vertices are roots of large trees and all have small excess, replace Δ by Δ/k. (The algorithm terminates when $\Delta < 1$.)

Rule 2. After a tree push from a vertex v to a vertex x, if the push is nonsaturating and $e(v) > 0$, push x onto S. Otherwise, while S is nonempty and its top vertex has small excess, pop S.

Rule 3. After relabeling a vertex, pop it from S, and continue to pop S until it is nonempty or its top vertex has large excess.

Every tree push made by the tree/scaling algorithm is from a vertex that is either the root of a small tree or has large excess. As discussed in Section 3, the time necessary for implementing Rules 1-3 is $O(n^2 + n \log_k (U + 1))$ plus $O(1)$ per tree push, which is dominated by the bound on tree pushes derived below.

Lemma 4.2. The tree/scaling algorithm makes $O(nm + kn^2 + \frac{n^2}{z} (\log_k U + 1))$ nonsaturating tree pushes.

Proof. The argument is analogous to the proof of Lemma 3.2. Let the potential Φ be defined by $\Phi = \sum_{v \text{ active}} e(v)\, d(v)/\Delta$. We have $0 \leq \Phi \leq n^2$. Increases in Φ are due to relabelings and changes in the value of Δ. The total increase in Φ due to relabelings is $O(n^2)$. Immediately after a change in the value of Δ, $\Phi \leq 2n^2/z$, because every active vertex must be the root of a large tree, and only $2n/z$ large trees can coexist. Thus the total increase in Φ caused by changes

in Δ is $O(\frac{n^2}{z}(\log_k U + 1))$.

Consider nonsaturating tree pushes that add a vertex to S. For such a push, let x be the vertex added to S. We charge the push either to the relabeling of x that pops x from S or to the at least $1-1/k$ decrease in Φ caused by pushes from x before x is popped. The total number of such pushes is thus $O(n^2 + \frac{n^2}{z}(\log_k U + 1))$. Similarly, the number of nonsaturating tree pushes that begin with more than one vertex on S and pop at least one vertex from S is also $O(n^2 + \frac{n^2}{z}(\log_k U + 1))$.

The last kind of nonsaturating tree push that we must count is a push from a vertex v that is the only vertex on S and is popped from S after the push. We call such a push an *emptying push*. An emptying push from a vertex v reduces $e(v)$ to zero. To count emptying pushes, we divide them into those from roots of small trees and those from roots of large trees.

We charge an emptying push from the root v of a small tree to the event that either created the tree or last made v active, whichever occurred first. If the tree was created after v became active, we charge the push to the link or cut that created the tree. Each link or cut is charged to at most one push, and there are $O(nm)$ links and cuts, which gives an $O(nm)$ bound on such pushes. If the tree was created before v became active, then v became active as a result of a saturating push, of which there are $O(nm)$, or as a result of a nonsaturating push from the root of a large tree. We have already counted the number of such pushes that are nonemptying. Thus it suffices to count emptying pushes from the roots of large trees.

We count such pushes using a second potential Φ_2. To define Φ_2, we need to define an integer l and a set P, as in the proof of Lemma 3.2. The value of l is the minimum of $2n$ and the smallest label of a large-excess root of a large tree added to S while S was empty during the current phase. (As in Section 3, we define a *phase* to consist of a maximal period of time during which Δ stays constant.) The set P consists of all vertices with label greater than l and all vertices with label equal to l from which an emptying push has been made during the current phase. The definition of Φ_2 is

$$\Phi_2 = \sum_{v \in P : e(v) > 0} e(v)\,(d(v) - l + 1)/\Delta. \quad \text{(If } P = \varnothing, \Phi_2 = 0.\text{)}$$

The total increase in Φ_2 over all phases is $O(n^2 + \frac{n^2}{kz}(\log_k U + 1))$. The $O(n^2)$ contribution is due to relabelings. The $O(\frac{n^2}{kz}(\log_k U + 1))$ contribution is due to decreases in l. A

decrease in l by one causes an $O(\frac{n}{kz})$ increase in Φ_2, since, just after such a decrease in l occurs, every vertex in P has level greater than l, and every active vertex in P has small excess and is the root of a large tree.

Every emptying push from the root of a large tree causes either a decrease of at least $1/k$ in Φ_2, which can happen $O(kn^2 + \frac{n^2}{z}(\log_k U + 1))$ times, or an addition of a vertex to P, which can happen $O(n(\log_k U + 1))$ times. Since $z \le n$, the latter bound is dominated by the former.

Combining all our estimates gives the lemma. □

Remark. A slightly weaker bound of $O(nm + kn^2 + \frac{kn^2}{z}(\log_2 U + 1))$ on the number of nonsaturating tree pushes can be obtained with a simpler argument using only Φ, analogous to the proof of Lemma 3.1. □

Theorem 4.3. With a choice of $k = \lceil \frac{m}{n} + 2 \rceil$ and $z = \min\{\lceil \frac{n}{m} \log_k U + 1 \rceil, n\}$ the tree/scaling preflow algorithm runs in $O(nm \log(\frac{n \log U}{m \log\log U} + 2))$ time.

Proof. By Lemmas 4.1 and 4.2, the running time of the algorithm is $O((nm + kn^2 + \frac{n^2}{z}(\log_k U + 1)) \log(z + 1))$. This is $O(nm \log n)$ if $z = \lceil \frac{n}{m} \log_k U + 1 \rceil < n$. On the other hand, if $z = n$ the computation is complete after one phase, and the running time of the algorithm is again $O(nm \log n)$. We consider two cases.

Case 1: $\log U / \log\log U > m/\sqrt{n}$. Then $\log(\frac{n \log U}{m \log\log U} + 2) = \Omega(\log n)$, and the theorem is true.

Case 2: $\log U / \log\log U \le m/\sqrt{n}$. Then $\log_2 U / \log_2\log_2 U \le m^{3/4}$, which implies $\log_2 U = O(m)$. Let us estimate the quantity $\frac{n^2}{z}(\log_k U + 1)$. If $z = n$, this quantity is $n(\log_k U + 1) = O(nm)$. On the other hand, if $z = \lceil \frac{n}{m} \log_k U + 1 \rceil$, $\frac{n^2}{z}(\log_k U + 1) \le 2nm$. It follows that the running time of the algorithm is $O(nm \log(z + 1)) = O(nm \log(\frac{n \log_k U}{m} + 2))$. It remains to prove that $\log(\frac{n \log_k U}{m} + 2) = O(\log(\frac{n \log U}{m \log\log U} + 2))$. We have $\log_k U = \log U / \log(\frac{m}{n} + 2)$.

Again the proof divides into two cases:

Case 2.1: $\log U > (\frac{m}{n} + 2)^4$. Then $\frac{1}{4} \log\log U > \log(\frac{m}{n} + 2) > \log(m/n)$. Also, $\log\log U > 4$, which implies $\log\log\log U \le \frac{1}{2} \log\log U$. Hence $\log(\frac{n \log U}{m \log\log U} + 2) \ge \log\log U - \log\log\log U - \log(m/n) \ge \frac{1}{4} \log\log U$. But then $\log(\frac{n \log_k U}{m} + 2) \le \log(\log U + 2) = O(\log(\frac{n \log U}{m \log\log U} + 2))$, as desired.

Case 2.2: $\log U \le (\frac{m}{n} + 2)^4$. In this case $\log\log U \le 4 \log(\frac{m}{n} + 2)$, which implies $\log(\frac{n \log_k U}{m} + 2) = \log(\frac{n \log U}{m \log(\frac{m}{n} + 2)} + 2) \le \log(\frac{4n \log U}{m \log\log U} + 2) = O(\frac{n \log U}{m \log\log U} + 2)$, as desired.

Thus in all cases the algorithm runs in the claimed time bound. □

Remark. The bound in the theorem can be improved to $O(nm \log(\frac{n \log U^*}{m \log\log U^*} + 2)$, analogously to the improvements in the bounds of Section 3. To obtain this improvement, we run one of the algorithms of Section 3 with $k = 2$ until $\Delta \le U^*$, and then switch to the dynamic tree algorithm. □

5. Remarks

The scaling preflow algorithm of Section 3 not only has a good theoretical time bound, but is likely to be very efficient in practice. We intend to conduct experiments to determine whether this is true. The algorithm of Section 4 is perhaps mainly of theoretical interest, although for huge networks there is some chance that the use of dynamic trees may be practical.

The two obvious open theoretical questions are whether further improvement in the time bound for the maximum flow problem is possible and whether our ideas extend to the minimum cost flow problem. It is not unreasonable to hope for a bound of $O(nm)$ for the maximum flow problem; note that the bounds of Theorems 3.3 and 4.3 are $O(nm)$ for graphs that are not too sparse and whose edge capacities are not too large, i.e. $\frac{\log U}{\log\log U} = O(m/n)$. Obtaining a bound better than $O(nm)$ would seem to require major new ideas.

As a partial answer to the second question, we have been able to obtain a time bound of $O(nm \log\log U \log(nC))$ for the minimum cost flow problem, where C is the maximum edge cost, assuming all edge costs are integral. This result uses ideas in the present paper and some additional ones. It will appear in a future paper.

6. Acknowledgements

We thank Andrew Goldberg for inspiring ideas and stimulating conversations.

7. References

[1] R.K. Ahuja and J.B. Orlin. "A fast and simple algorithm for the maximum flow problem." Technical Report 1905-87, Sloan School of Management, M.I.T., 1987.

[2] J. Cheriyan and S.N. Maheshwari. "Analysis of preflow push algorithms for maximum network flow." Department of Computer Science and Engineering, Indian Institute of Technology, New Delhi, India, 1987.

[3] J. Edmonds and R.M. Karp. "Theoretical improvements in algorithmic efficiency for network flow problems." *Journal of the ACM*, 19:248-264, 1972.

[4] S. Even. *Graph Algorithms*. Computer Science Press, Potomac, MD, 1979.

[5] R.L. Ford, Jr. and D.R. Fulkerson. *Flows in Networks*. Princeton University Press, Princeton, NJ, 1962.

[6] H.N. Gabow. "Scaling algorithms for network problems." *J. of Comp. and Sys. Sci.*, 31:148-168, 1985.

[7] A.V. Goldberg. "Efficient Graph Algorithms for Sequential and Parallel Computers." Ph.D. Thesis, M.I.T., 1987.

[8] A.V. Goldberg. "A New Max-Flow Algorithm." Technical Report MIT/LCS/TM-291, Laboratory for Computer Science, M.I.T., 1985.

[9] A.V. Goldberg and R.E. Tarjan. "A new approach to the maximum flow problem." In *Proc. 18th ACM Symp. on Theory of Computing*, pages 136-146, 1986; *J. Assoc. Comput. Mach.*, to appear.

[10] A.V. Karzanov. "Determining the maximal flow in a network by the method of preflows." *Soviet Math. Dokl.*, 15:434-437, 1974.

[11] E.L. Lawler. *Combinatorial Optimization: Networks and Matroids*. Holt, Reinhart, and Winston, New York, NY, 1976.

[12] V.M. Malhotra, M. Pramodh Kumar, and S.N. Maheshwari. "An $O(|V|^3)$ algorithm for finding maximum flows in networks." *Inform. Process. Lett.*, 7:277-278, 1978.

[13] C.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-hall, Englewood Cliffs, NJ, 1982.

[14] Y. Shiloach and U. Vishkin. "An $O(n^2 \log n)$ parallel max-flow algorithm." *Journal of Algorithms*, 3:128-146, 1982.

[15] D.D. Sleator. "An $O(nm \log n)$ Algorithm for Maximum Network Flow." Technical Report STAN-CS-80-831, Computer Science Department, Stanford University, 1980.

[16] D.D. Sleator and R.E. Tarjan. "A data structure for dynamic trees." *J. Comput. System Sci.*, 26:362-391, 1983.

[17] D.D. Sleator and R.E. Tarjan. "Self-adjusting binary search trees." *J. Assoc. Comput. Mach.* 32:652-686, 1985.

[18] R.E. Tarjan. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[19] R.E. Tarjan. "A simple version of Karzanov's blocking flow algorithm." *Operations Research Letters* 2:265-268, 1984.

END
DATE
FILMED
7-88
DTIC